Java 8 Programmer II Study Guide: Exame 1Z0-809

Parte UM

Design de Classe

Capítulo UM

Encapsulamento e Classes Imutáveis

Objetivos do Exame

- Implementar encapsulamento.
- Criar e usar classes singleton e imutáveis.
- Desenvolver código que usa a palavra-chave static em blocos de inicialização, variáveis, métodos e classes.
- Desenvolver código que usa a palavra-chave final.

Encapsulamento

Como sabemos, Java é uma linguagem orientada a objetos, e todo o código deve estar dentro de uma classe:

```java
class MyClass {
    String myField = "you";
    void myMethod() {
        System.out.println("Hi " + myField);
    }
}
```

Portanto, temos que criar um objeto para usá-lo:

```java
MyClass myClass = new MyClass();
myClass.myField = "James";
myClass.myMethod();
```

Um conceito importante em linguagens de programação orientadas a objetos é o encapsulamento, a habilidade de esconder ou proteger os dados de um objeto.

Na maioria das vezes, quando alguém fala sobre encapsulamento, as pessoas tendem a pensar em variáveis privadas e métodos getters e setters públicos, e em como isso é excessivo. Mas na verdade, encapsulamento é mais do que isso, e é útil para criar designs de alta qualidade.

Vamos considerar o exemplo anterior.

Antes de tudo, é bom esconder o máximo possível a implementação interna de uma classe.
A razão: mitigar o impacto das mudanças.

Por exemplo, e se myField mudar seu tipo de String para int?

```java
myClass.myField = 1;
```

Se estivéssemos usando esse atributo em cinquenta classes, teríamos que fazer cinquenta mudanças em nosso programa.

Mas se escondêssemos o atributo e usássemos um método setter em vez disso, poderíamos evitar as cinquenta mudanças:

```java
// Escondendo o atributo do mundo externo com a palavra-chave private
private int myField = 0;

void setMyField(String val) { // Ainda aceitando uma String
    try {
        myField = Integer.parseInt(val);
    } catch(NumberFormatException e) {
        myField = 0;
    }
}
```

Você implementa essa ocultação de atributos usando modificadores de acesso.

Java suporta quatro modificadores de acesso:

- public
- private
- protected
- default (quando nenhum modificador é especificado)

Você pode aplicar esses modificadores a classes, atributos e métodos de acordo com a seguinte tabela:

Classe/Interface	Classe	Interface	Atributo	Método	Atributo	Método
`public`	X	X	X	X	X	
`private`			X	X		
`protected`			X	X		
`default`	X		X	X	X	

Como você pode ver, todos os modificadores de acesso podem ser aplicados a atributos e métodos de uma classe, mas não necessariamente aos seus equivalentes em interfaces. Além disso, definições de classe e interface só podem ter modificadores public ou default.
Por quê? Vamos definir primeiro esses modificadores de acesso.

- Se algo é declarado como public, pode ser acessado de qualquer outra classe em nossa aplicação, independentemente do pacote ou módulo onde está definido.
- Se algo é definido como private, só pode ser acessado dentro da classe que o define, não por outras classes no mesmo pacote e nem por classes que herdam dessa classe. private é o modificador de acesso mais restritivo.
- Se algo é definido como protected, só pode ser acessado pela classe que o define, suas subclasses e classes do mesmo pacote. Não importa se a subclasse está em outro pacote, o que torna esse modificador menos restritivo que private.

Se algo não tem modificador, tem acesso default, também conhecido como acesso de pacote, pois só pode ser acessado por classes dentro do mesmo pacote. Se uma subclasse for definida em outro pacote, ela não poderá ver o atributo ou método com acesso default. Essa é a única diferença em relação ao modificador protected, o que o torna mais restritivo.

Um exemplo de código pode explicar isso melhor:

```java
package street21;
public class House {
    protected int number;
    private String reference;

    void printNumber() {
        System.out.println("Num: " + number);
    }
    public void printInformation() {
        System.out.println("Num: " + number);
        System.out.println("Ref: " + reference);
    }
}
class BlueHouse extends House {
    public String getColor() { return "BLUE"; }
}
```

```java
package city;
import street21.*;

class GenericHouse extends House {
    public static void main(String args[]) {
        GenericHouse h = new GenericHouse();
        h.number = 100;
        h.reference = ""; // Erro de compilação
        h.printNumber(); // Erro de compilação
        h.printInformation();
        BlueHouse bh = new BlueHouse(); // Erro de compilação
        bh.getColor(); // Erro de compilação
    }
}
```

- h.number compila porque este atributo é protected e GenericHouse pode acessá-lo por estender House.
- h.reference não compila porque este atributo é private.
- h.printNumber() não compila porque este método tem acesso default (de pacote).
- h.printInformation() compila porque este método é public.
- BlueHouse bh = new BlueHouse() não compila porque a classe tem acesso default.
- bh.getColor() não compila porque, embora o método seja public, a classe que o contém não é.

Agora, consegue ver por que alguns modificadores se aplicam a certos elementos e outros não?

Faria sentido uma classe private ou protected em uma linguagem orientada a objetos?
E um método private dentro de uma interface onde, na maioria das vezes, os métodos não têm implementação?

Pense sobre isso.

Resumo das regras:

Membros	Mesma classe	Subclasse mesmo pacote	Subclasse pacote diferente	Outra classe mesmo pacote	Outra classe pacote diferente
`public`	✓	✓	✓	✓	✓
`private`	✓				
`protected`	✓	✓	✓	✓	
`default`	✓	✓		✓	

O que é um Singleton?

Haverá momentos em que você desejará ter apenas uma instância para uma determinada classe. Tal classe é uma classe singleton e é um padrão de projeto.

Algumas classes do Java são escritas usando o padrão singleton, por exemplo:

- java.lang.Runtime
- java.awt.Desktop

Em Java, de maneira geral, há duas formas de implementar uma classe singleton:

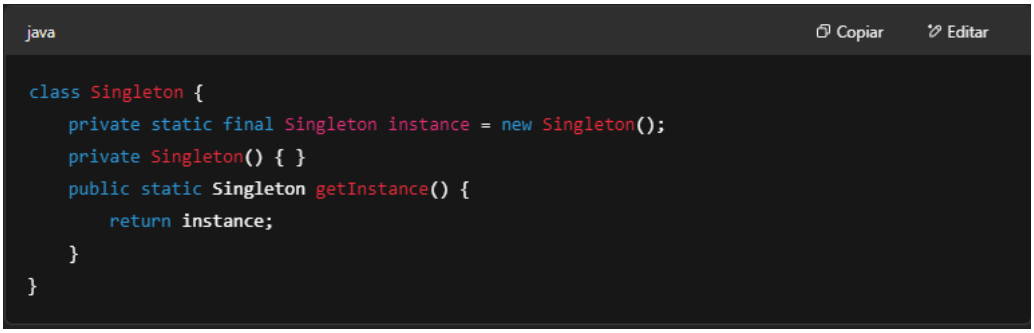
- Com um construtor private e um método de fábrica static
- Como um enum

Vamos começar com o modo do construtor private. Embora possa parecer simples no início, ele apresenta muitos desafios, todos relacionados a manter apenas uma instância da classe singleton durante todo o ciclo de vida da aplicação.

Ao ter um construtor private, a classe singleton garante que nenhuma outra classe crie uma instância dela. Um método private (ou, neste caso, o construtor) só pode ser usado dentro da própria classe que o define.

Assim, a instância tem que ser criada dentro da própria classe.

Isso pode ser feito de duas maneiras. Usando um atributo static private e um método para obtê-lo:



- O atributo deve ser private, para que nenhuma outra classe possa usá-lo, apenas via o getter.
- Deve ser static, para que a instância seja criada quando a classe for carregada antes de alguém usá-la e porque membros static pertencem à classe e não a uma instância específica.
- E deve ser final, para que uma nova instância não possa ser criada depois.

Uma variação dessa implementação é usar uma classe interna static (veremos esse tipo de classe no Capítulo 3):

```java
class Singleton {
    private Singleton() { }
    private static class Holder {
        private static final Singleton instance = new Singleton();
    }
    public static Singleton getInstance() {
        return Holder.instance;
    }
}
```

A vantagem disso é que a instância não será criada até que a classe interna seja referenciada pela primeira vez.

Contudo, haverá momentos em que, por exemplo, criar o objeto não é tão simples quanto chamar new, então a outra forma é criar a instância dentro do método get:

```java
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
            // mais código para criar a instância...
        }
        return instance;
    }
}
```

Na primeira vez que este método for chamado, a instância será criada. Mas com essa abordagem, enfrentamos outro problema. Em um ambiente com múltiplas threads, se duas ou mais threads executarem este método em paralelo, há um risco significativo de acabar com várias instâncias da classe.

Uma solução é sincronizar o método para que apenas uma thread por vez possa acessá-lo.

```java
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

O problema com isso é que, estritamente falando, não é muito eficiente. Precisamos de sincronização apenas na primeira vez, quando a instância é criada, não todas as vezes que o método é chamado.

Uma melhoria é bloquear apenas a parte do código que cria a instância. Para isso funcionar corretamente, precisamos verificar se a instância é null duas vezes — uma sem o bloqueio (para ver se já foi criada), e outra dentro do bloco synchronized (para criá-la com segurança):

```java
class Singleton {
    private static Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Mas, novamente, essa implementação ainda não é perfeita. O problema agora está no nível da Máquina Virtual Java (JVM). A JVM, ou às vezes o compilador, pode otimizar o código reordenando ou armazenando em cache o valor de variáveis (e não tornando as atualizações visíveis).

A solução é usar a palavra-chave volatile, que garante que qualquer operação de leitura/gravação de uma variável compartilhada por várias threads seja atômica e não armazenada em cache:

```java
class Singleton {
    private static volatile Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Como você pode ver, dá trabalho implementar corretamente um singleton quando você deseja ou precisa adiar a instanciação da classe até que ela seja usada pela primeira vez (também chamado de lazy initialization). E não cobriremos serialização nem como manter um singleton em um cluster.

Então, se você não precisa disso, use os dois primeiros métodos (criar a instância na declaração da variável ou usar a classe interna Holder) ou a forma mais fácil (e recomendada): uma enumeração (enum).

Veremos enums no Capítulo 5, por enquanto, basta saber que enums são singletons.

Objetos Imutáveis

Haverá outros momentos em que você não desejará modificar os valores ou o estado de um objeto quando utilizado por várias classes. Tal objeto será um objeto imutável.

Existem algumas classes imutáveis no JDK do Java, por exemplo:

- java.lang.String
- Classes *Wrapper* (como Integer)

Objetos imutáveis **não podem mudar depois de serem criados**. Isso significa que eles não podem ter métodos setter ou variáveis públicas. A única forma de definir suas propriedades é através do construtor.

Imutabilidade também significa que, se um método precisar alterar as propriedades do objeto, como o objeto ou seus valores não podem mudar, ele precisa retornar uma cópia do objeto com os novos valores (é assim que a classe String funciona).

Outro ponto a considerar é a herança. Se a classe imutável puder ser herdada, a subclasse pode alterar os métodos para modificar as instâncias da classe. Assim, uma classe imutável **não pode permitir isso**.

Em resumo, um objeto imutável:

- Define todas as suas propriedades através de um construtor
- Não define métodos setter
- Declara todos os seus atributos como private (e às vezes final)
- Tem sua classe declarada como final para prevenir herança
- Protege o acesso a qualquer estado mutável. Por exemplo, se tiver um membro List, ou a referência não pode ser acessível externamente, ou deve retornar uma cópia (o mesmo se aplica caso o conteúdo do objeto precise mudar)

Palavra-chave static

Pense em algo static como algo pertencente à classe, e não a uma instância específica dessa classe.

Se estamos falando de atributos, um atributo static é compartilhado entre todas as instâncias da classe (porque, novamente, não pertence a uma instância).

Compare a saída deste código:

java Copiar Editar

```java
public class Example {
    private int attr = 0;

    public void printAttr() {
        System.out.println(attr);
    }

    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        Example e2 = new Example();
        e2.printAttr();
    }
}
```

Saída:

Copiar Editar

```
10
0
```

Com a saída do código que usa uma variável static:

```java
public class Example {
    private static int attr = 0;

    public void printAttr() {
        System.out.println(attr);
    }

    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        Example e2 = new Example();
        e2.printAttr();
    }
}
```

Saída:

```
10
10
```

Como você pode ver, o valor é retido quando outra instância usa a variável static.

Falando de métodos static, o método pertence à classe, o que significa que **não precisamos de uma instância para chamá-lo** (o mesmo se aplica aos atributos).

```java
public class Example {
    private static int attr = 0;

    public static void printAttr() {
        System.out.println(attr);
    }

    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        // Referenciando o método de forma estática
        Example.printAttr();
    }
}
```

Saída:

```
10
10
```

Entretanto, se observar atentamente, printAttr usa um atributo static, e esse é o ponto com métodos static: eles **não podem usar variáveis de instância**, apenas variáveis static.

Isso faz sentido: se você pode acessar um método static apenas com a classe, não há garantia de que exista uma instância, e mesmo que exista, como associar um atributo à instância quando você tem apenas o nome da classe?

Pelo mesmo motivo, as palavras-chave super e this **não podem ser usadas**.

Classes static serão abordadas no Capítulo 3, mas há outra construção que pode ser marcada como static: um bloco de inicialização.

Bloco de Inicialização static

```java
public class Example {
    private static int number;

    static {
        number = 100;
    }
}
```

Um bloco é executado quando a classe é inicializada e **na ordem em que são declarados** (se houver mais de um).

Assim como métodos static, eles não podem fazer referência a atributos de instância, nem usar as palavras-chave super e this.

Além disso, blocos static não podem conter uma instrução return, e ocorre erro de compilação se o bloco não puder ser concluído normalmente (por exemplo, devido a uma exceção não capturada).

Palavra-chave final

A palavra-chave final pode ser aplicada a variáveis, métodos e classes.

- Quando final é aplicado a **variáveis**, você **não pode alterar o valor** da variável após sua inicialização. Essas variáveis podem ser atributos (static ou não) ou parâmetros.
- Atributos final podem ser inicializados:
    - Na declaração
    - Dentro de um construtor
    - Dentro de um bloco de inicialização

```java
public class Example {
    private final int number = 0;
    private final String name;
    private final int total;
    private final int id;

    {
        name = "Name";
    }

    public Example() {
        number = 1; // Erro de compilação
        total = 10;
    }

    public void exampleMethod(final int id) {
        id = 5; // Erro de compilação
        this.id = id; // Erro de compilação
    }
}
```

- Quando final é aplicado a um **método**, este **não pode ser sobrescrito**.

```java
public class Human {
    final public void talk() { }
    public void eat() { }
    public void sleep() { }
}

public class Woman extends Human {
    public void talk() { } // Erro de compilação
    public void eat() { }
    public void sleep() { }
}
```

- Quando final é aplicado a uma **classe**, ela **não pode ser estendida**.

```java
public final class Human {
    ...
}

public class Woman extends Human { // Erro de compilação
    ...
}
```

Em resumo:

- Uma variável final só pode ser inicializada uma vez e não pode ter seu valor alterado depois.
- Um método final não pode ser sobrescrito por subclasses.
- Uma classe final não pode ser estendida.

Pontos-Chave

- Encapsulamento é a capacidade de esconder ou proteger os dados de um objeto.
- Java suporta quatro modificadores de acesso: public, private, protected, e default (quando nenhum é especificado, também chamado de nível de pacote).
- Se algo é declarado como public, pode ser acessado de qualquer outra classe da aplicação, independentemente do pacote ou módulo onde está definido.
- Se algo é definido como private, só pode ser acessado dentro da classe que o define — não por outras classes no mesmo pacote nem por classes que a herdam. private é o modificador de acesso mais restritivo.
- Se algo é definido como protected, pode ser acessado pela classe que o define, por suas subclasses e por classes do mesmo pacote — mesmo que a subclasse esteja em outro pacote.
- Se algo não tem modificador, tem acesso padrão (default, ou "acesso de pacote"), acessível apenas por classes dentro do mesmo pacote.
- Uma classe singleton garante que há apenas uma instância da classe durante a vida da aplicação.
- Um objeto imutável não pode ter seus valores alterados após ser criado.

Um objeto imutável:

- Define todas as suas propriedades através de um construtor
- Não define métodos setter

- Declara todos os seus atributos como private (e às vezes final)
- Tem a classe declarada como final para evitar herança
- Protege o acesso a qualquer estado mutável (por exemplo, se tem uma List, a referência não pode ser exposta ou uma cópia deve ser retornada)
- A palavra-chave static pode ser aplicada a atributos, métodos, blocos e classes.
- Um membro static pertence à classe onde é declarado, e não a uma instância específica.
- A palavra-chave final pode ser aplicada a variáveis (para impedir mudanças após a inicialização), métodos (para evitar sobrescrita) e classes (para impedir herança).

Autoavaliação

1. Dado:

```java
public class Question_1_1 {
    private final int a; // 1
    static final int b = 1; // 2
    public Question_1_1() {
        System.out.print(a); // 3
        System.out.print(b); // 4
    }
}
```

Qual é o resultado?

A. 01
B. A compilação falha na linha marcada como // 1
C. A compilação falha na linha marcada como // 2
D. A compilação falha na linha marcada como // 3
E. A compilação falha na linha marcada como // 4

2. Qual das seguintes alternativas apresenta a ordem correta do modificador mais restritivo para o menos restritivo?

A. private, default, public, protected
B. protected, private, default, public
C. default, protected, private, public
D. private, default, protected, public

3. Dado:

```java
public final class Question_1_3 {
    private final int n;
    public Question_1_3() { }
    public void setN(int n) { this.n = n; }
}
```

Qual das seguintes alternativas é verdadeira?

A. A classe é imutável
B. A classe não é imutável

C. A compilação falha
D. Uma exceção ocorre em tempo de execução

4. Dado:

```java
public class Question_1_4 {
    private final List<Integer> list = new ArrayList<>();
    public void add() {
        list.add(0);
    }
    public static void main(String[] args) {
        Question_1_4 q = new Question_1_4();
        q.add();
    }
}
```

Qual das seguintes alternativas é verdadeira?

A. O atributo list contém um elemento após a execução do main
B. A classe é imutável
C. A compilação falha
D. Uma exceção ocorre em tempo de execução

5. Dado:

```java
public class Question_1_5 {
    private String s = "Hi";

    public static void main(String[] args) {
        Question_1_5 q = new Question_1_5();
        q.s = "Bye"; // 1
        System.out.println(q.s); // 2
    }
}
```

Qual é o resultado?

A. Hi
B. Bye
C. A compilação falha na linha marcada como // 1
D. A compilação falha na linha marcada como // 2

6. Dado:

```java
public class Question_1_6 {
    private static int a;
    private int b;
    static {
        a = 1;
        b = 2;
    }
    public static void main(String[] args) {
        Question_1_6 q1 = new Question_1_6();
        Question_1_6 q2 = new Question_1_6();
        q2.b = 1;
        System.out.println(q1.a + q2.b);
    }
}
```

Qual é o resultado?

A. 0
B. 3
C. 2
D. A compilação falha